femington
Int. J. Intell. Syst. Data Sci. (2026) 1(1):1–12
Article ID: IJISDS-2026-150

Evaluating the Effectiveness of the Mining Android Sandbox (MAS) Approach for Malicious App Detection in Large Datasets

Apeksha Bhuekar
apeksharaj17@gmail.com
Independent Researcher

Abstract

Users widely favor android smartphones for their valued and ever-growing services. As a result, there has been a wide usage of smartphones by a host of users. For instance, a mobile phone device like a smartphone is quite popular among kids, youngsters, and adults too for their day-to-day activities. Increased utilization of smartphones is causing security issues also. Because of the enhancement of malware through virus kits, today the security path has become really difficult for the users. These malware kits enhance the capabilities of malware coders, providing a set of customizable features for ease of use. In simple terms, a virus kit can quickly produce new malware variants based on the previous malware. It is imperative that we see how malware and MBR-Virus Kits have evolved over the years. The designer always creates the malware on the basis of the malware already designed. A significantly high false alarm rate is one of the issues the detection system is encountering which deteriorates the detection system's working. Only use a metal detector when you can realize that it produces a low false alarm in situations where relevant data has strong confirming evidential support. It is not enough for most dubious scenarios which is why uncertainty analysis and measurement are essential for designing better detection systems.

Keywords

• Android Malware Detection • Repackaged Application Analysis • Dynamic Analysis and Sandbox Mining • Mining Android Sandbox (MAS) • Secure Malware Family Classification

1. Introduction

Mobile technologies are ubiquitous, with Android holding over 70% market share and hosting nearly 2.5 million apps on Google Play (June 2023). This scale increases security risks, motivating efforts to detect malicious or vulnerable Android apps. A common threat is repackaging, where benign apps are modified with malicious code (e.g., data exfiltration [1]) and redistributed. The Mining Android Sandbox (MAS) detects such behavior in two phases: an exploratory phase that records sensitive API usage via automated testing, and an enforcement phase that flags unseen sensitive API calls during execution.Prior studies [2, 3] show MAS, especially with DroidBot [4], effectively detects repackaged malware. However, they rely on a small dataset (102 pairs) and do not analyze how factors like malware presence, app similarity, or malware family affect accuracy. Recent studies have explored machine learning and hybrid analysis techniques for Android malware detection, reporting high accuracy under controlled settings, but often struggling with repackaged malware and real-world variability.

© 2026 Femington Academic Publishing. All rights reserved.

We address these gaps through three research questions: (1) how MAS accuracy scales to larger datasets, (2) how app similarity impacts performance, and (3) how malware families influence detection. Using a substantially larger dataset (4,076 pairs), we reevaluate MAS with DroidBot, detailing methodology in Section 3.

Negative results. We observe a significant drop in performance ($F_1 = 0.54$ vs. 0.89 previously). Analysis shows MAS struggles with certain families, particularly Gappusin: 1,170 of 1,337 samples are misclassified, substantially reducing recall. Results, discussion, and threats to validity are presented in Sections 4 and 5, with conclusions in Section 6.

2. Background and Related Work

Android bytecode can be easily reverse engineered, enabling attackers to decompile legitimate apps, inject malicious code, and redistribute them. These repackaged apps exploit the popularity of trusted apps to spread malware [1]. For instance, a repackaged Pokémon Go app appeared within 72 hours of its 2016 release, gaining full device access. Repackaging remains a significant threat, estimated to affect about 25% of Google Play apps, while detection mechanisms are still limited. This endangers user privacy and violates developer rights, motivating analysis techniques such as MAS for malware detection [2, 5]

2.1 Extraction of Android Sandboxes

A *sandbox* isolates applications to prevent unauthorized access to system resources, enforcing the principle of least privilege. In Android, access to sensitive resources (e.g., contacts, camera) is controlled via sensitive APIs and permissions. The MAS [5] builds such sandboxes through dynamic analysis. It observes sensitive API calls during an exploratory phase, derives rules from them, and blocks unseen calls during execution. Implementations like Boxmate extend this by recording (event, API) pairs, enabling finer-grained control. MAS can combine static and dynamic analysis by instrumenting apps and using test generators (e.g., DroidMate, DroidBot, Monkey) to collect runtime API usage. Higher code coverage generally leads to more accurate sandboxes.

Why DroidBot? DroidBot is used due to its state-aware, UI-guided exploration, achieving higher coverage and better malware detection accuracy than alternatives like Monkey or DroidMate.

2.2 Malware Classification through Android Sandbox Mining

Beyond generating sandboxes, MAS effectively detects malicious behavior in repackaged Android apps [2]. Effectiveness is accuracy in correctly identifying malicious behavior in repackaged versions. MAS for malware classification typically works as follows:

1. **Instrumentation**: instrument original and repackaged apps.
2. **Exploration**: collect set S_1 of sensitive API calls from the original app while running a test generator (e.g., DroidBot).
3. **Exploitation**: collect set S_2 from the repackaged app, compute $S = S_2 \setminus S_1$, and classify as malware if $|S| > 0$.

Prior studies evaluated MAS effectiveness on repackaged malware. Bao et al. [2]. According to their report, sandboxes built with test generators classify at least 66% of 102 pairs of repackaged apps as malware. Their larger experiment (102 pairs, 1 minute per tool) is the one we replicate. Among the evaluated tools, DroidBot [4] produced the most effective sandbox. Subsequent work extended MAS with parameter-value checks (Le et al.) and combined it with static analysis (Costa et al.), with DroidFax classifying nearly half of repackaged apps as malware.

The MAS classification algorithm compares sensitive API call sets. It instruments apps with DroidFax, executes each via DroidBot, and compares observed call sets. The decision rule:

$$\text{If } |S_2 \setminus S_1| > 0 \Rightarrow \text{malware}$$

2.3 Classification of Android Malware – Contemporary

Most recent surveys on Android malware detection categorize approaches into static, dynamic, and hybrid techniques. Dynamic analysis observes runtime behavior, capturing threats missed statically, but often suffers from limited code coverage and higher computational cost. It can run on emulators or real devices, with the choice affecting detection since some malware evades emulated environments. Hybrid approaches aim to improve coverage and robustness.

Machine learning methods use static and dynamic features to achieve high reported accuracy (¿90%) and detect unseen families. However, their effectiveness depends heavily on dataset quality, feature selection, and representativeness. They are also vulnerable to adversarial behavior that manipulates model input. Beyond MAS, tools like FlowDroid, DroidSIFT, and Drebin have been evaluated—while Drebin reaches 93% accuracy on DEX features, it struggles with repackaged apps. MAS, in contrast, is more resilient to minor obfuscations but less effective for JNI and native code. Recent surveys also highlight that repackaging-based attacks remain challenging for existing detection techniques, especially under high similarity conditions.

3. Experimental Setup

Everyone is aware of the malware attacks that have become a daily thing. Almost, every day malware attacks take place. A variety of motives, including money, espionage, and hacktivism, motivates malicious actors. There are different types of malware that might cause serious damage to your computer system that includes virus, worm, spyware, etc [2, 3]. In this part, we describe the study settings. To begin, we present the ways through which we create our datasets. (Section 3.1). TNext, we discuss the data gathering and analytical processes (Sections 3.2 and 3.3).

3.1 Malicious Software Dataset

The set of data used is capable to answer the problem statement. It consists of a large number of repackaged Android apps, across a diverse set of malware families, along with correct labels for important attributes, including similarity and malware family.

3.1.1 Dataset Construction Procedures.

The dataset construction is performed in three phases that are summarized in Table 1.

Table 1: Sequential filtering steps to construct the final dataset.

Step	Description	Remaining Pairs
Initial	Raw pairs from RePack and AMC	16,487
1	Remove original apps that failed instrumentation	5,875
2	Remove repackaged apps that failed instrumentation	4,742
3	Remove apps that could not be installed on emulator	4,737
4	Remove apps flagged as malware by VirusTotal	4,076

First, we constructed our dataset using two repackaged Android app repositories: RePack [1] and AMC. RePack, which is derived from Androzoo [6], includes 18,073 applications (2,776 original and 15,297 repackaged), with multiple repackaged variants for each original app. It is widely used in repackaging research. However, its coverage is limited to apps developed up to 2018. To address this, we supplemented it with more recent samples from AMC. Since AMC does not provide information about original apps, we applied a previously proposed heuristic [1] to identify them, which resulted in 1,190 post-2018 pairs. In total, the initial dataset consisted of 16,487 app pairs.

Next, during experimentation, several issues arose with a significant number of samples, particularly during DroidFax instrumentation and execution on the Android emulator. We were unable to instrument 919 original apps (from both RePack and AMC), which reduced the dataset to 5,875 pairs. Among these, 430 repackaged apps also failed during instrumentation. In addition, 586 failures occurred during analysis or log generation, bringing the total down to 4,742 pairs. Finally, five apps could not be installed on the Android emulator (API level 28). Similar challenges have been reported in earlier studies, often with even higher failure rates [2].

Finally, we used VirusTotal (VT) to detect any original apps flagged as malware. Since MAS assumes that original apps are benign, any such samples were removed to avoid inconsistencies, as their repackaged versions could also be malicious. VT scans applications using more than 60 antivirus engines. Based on this step, 661 samples were excluded. After all filtering steps, the final dataset (LargeDS) contains 4,076 app pairs. For reproducibility, we also include the smaller dataset (SmallDS) used in previous studies [2, 3].

3.1.2 Characteristics of the Datasets

In our examination of repackaged malware, we asked VT for all items reported as malicious apps. Out of the 102 app pairs present in SmallDS, we find that 69 pairs (67.64%) were reported by at least two SEs. We take this as an indication of severity threshold malware, based on prior work's normalisation against CT. [7]. In LargeDS, 2,895 of 4,076 apps (71.01%) meet this criterion, with consistent trends observed under alternative thresholds (≥ 1, ≥ 5, ≥ 10 SEs; Section 5).

Malware spans categories such as riskware, trojan, and adware, and can be grouped into families based on attack strategies. Using Avast Threat Intelligence (AVT), SmallDS covers 17 families, mainly Kuguo (49.27%) and Dowgin (17.39%), while LargeDS includes 116 families, dominated by Gappusin (46.18%). VT could not assign families to 253 LargeDS samples. SimiDroid helps determine how similar the original app and a clone. LargeDS shows an average similarity of 90.39% (87 pairs < 25%, 49 pairs 25–50%, 353 pairs 50–75%, 3,587 pairs > 75%), while SmallDS averages 89.41%.

Across both datasets, repackaged apps injected 133 distinct sensitive APIs (AppGuard [8]), enabling data access and other malicious actions. Table 2 lists the 10 most frequent APIs in LargeDS.

Expanding MAS's sensitive API list to include native method invocations (e.g., 'System.loadLibrary')

and dynamic class loading calls ('DexClassLoader') may improve detection. These calls are frequently used in advanced malware but are currently excluded from AppGuard's list. It is worth emphasizing that the samples were obtained from various Android app stores. Majority of our repackaged apps are sourced from the unofficial android app store Anzhi. The Google Play Store, or Google Play, is a digital distribution platform for Android devices. Excluding apps that failed instrumentation or were flagged by VirusTotal may introduce selection bias, but ensures alignment with MAS assumptions and data quality. Potential under-representation is mitigated by using a large, diverse dataset.

Table 2: Sensitive APIs Frequently Used in Repackaged Apps. The *Occurrences* column indicates the number of distinct repackaged apps introducing calls to sensitive methods.

Sensitive API Method	Occurrences
`android.telephony.TelephonyManager: int getPhoneType()`	311
`android.telephony.TelephonyManager: String getNetworkOperatorName()`	297
`android.location.LocationManager: String getBestProvider(Criteria, boolean)`	292
`android.telephony.TelephonyManager: int getSimState()`	284
`java.lang.reflect.Field: Object get(Object)`	277
`android.net.NetworkInfo: String getTypeName()`	271
`android.database.sqlite.SQLiteDatabase: Cursor query(String, String[], ...)`	270
`java.lang.reflect.Field: int getInt(Object)`	250
`android.net.wifi.WifiInfo: String getMacAddress()`	238
`android.telephony.TelephonyManager: String getNetworkOperator()`	237

3.2 Data Collection Procedures

We used DroidXP [9] for data collection. We only used DroidXP to automate our steps and not for the comparison of test case generation tools for malicious behaviour identification in MAS.

1. **Instrumentation**: We used DroidXP to instrument all app pairs (original and repackaged), leveraging DroidFax for instrumentation and static analysis. This step is performed once per app version.

2. **Execution**: DroidXP runs instrumented apps on an Android emulator (API 28) using DroidBot [4]. Each app is executed three times for three minutes to improve coverage and reduce randomness, with emulator data reset between runs.

3. **Data Collection**: DroidXP then uses DroidFax to gather execution data (e.g., sensitive API calls, coverage metrics) for MAS performance analysis.

Instrumentation impacts detection accuracy by determining which API calls are monitored. Missing reflective or native calls (common in obfuscated malware) can hide malicious behavior. In our study, such failures led to excluding 919 app pairs, affecting results. Figure 1 shows the overall MAS architecture.

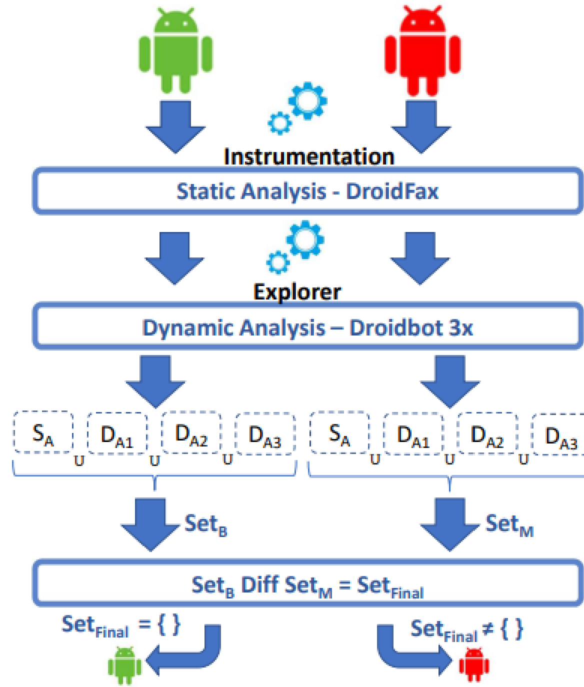


Figure 1: The overall architecture of the MAS

Let S_1 be the sensitive API calls from the original app, and S_2 from the repackaged version. The difference set is:

$$S_{\text{diff}} = S_2 \setminus S_1$$

A repackaged app is labeled malicious if $|S_{\text{diff}}| > 0$.

DroidXP logs code coverage metrics (executed methods and lines) per run. We report average line and method coverage over three independent executions to mitigate randomness and emulator noise.

3.3 Data Analysis Process

We use DroidBot as a sandbox environment to determine whether a repackaged app should be labeled as malware[10]. An app is marked as malicious if it calls at least one sensitive API that appears only in the repackaged version and not in its original counterpart. If no such API usage is found, the app is not considered malware. The set of sensitive APIs is based on AppGuard [8], following the mappings provided by Song et al. To strengthen our analysis, we further compare the MAS results with VirusTotal, which leads to the following cases:

- **True Positive (TP)**: The MAS flags a repackaged app as malware, and VirusTotal (VT) also reports it as malicious, with at least two security engines (SEs) detecting it.
- **True Negative (TN)**: The MAS does not flag the repackaged app as malware, and VT shows little to no concern, with at most one SE marking it as malicious.

- **False Positive (FP)**: The MAS classifies the repackaged app as malware, but VT provides weak support for this decision, with at most one SE identifying it as malicious.
- **False Negative (FN)**: The MAS does not detect the repackaged app as malware, even though VT indicates otherwise, with at least two SEs labeling it as malicious.

In Section 4, we measure Precision, Recall, and F_1 using the standard definitions of true and false positives and negatives. To summarize performance, we report the mean, median, and standard deviation for both SmallDS (102 pairs) and LargeDS (4,076 pairs). We also study how app similarity influences MAS performance through Spearman correlation and logistic regression. Finally, we examine a subset of repackaged apps through reverse engineering to identify the causes of misclassification.

4. Results

In the forthcoming segment, we present our results. It is important to remind the reader about the aim of this study, which is to understand the advantages and shortcomings of MAS for malware detection using state-of-the-art test case generation (DroidBot)[11]. Our analysis is performed on two datasets, smallDS which contains 102 pairs of apps and largeDS which contains 4,076 pairs of apps.

4.1 Exploratory Data Analysis of Accuracy

On SmallDS (102 apps), the MAS classifies 69 repackaged apps as malware (67.64%), closely matching Bao et al.'s 66.66% [2], confirming reproducibility. Prior work [2, 3] assumes all repackaged apps are malicious and thus does not report accuracy metrics. In contrast, we use VT to build ground truth, labeling an app as malware if at least two security engines flag it [7]. Under this setup, the MAS achieves 0.89 accuracy on SmallDS, with 7 false negatives and 7 false positives.

Table 3: Accuracy of the MAS on the SmallDS (102 pairs) and LargeDS (4,076 pairs). Precision, recall, and F1-score are reported.

Dataset	TP	FP	FN	Precision	Recall	F_1
SmallDS (102)	62	7	7	0.89	0.89	0.89
LargeDS (4,076)	1,175	220	1,720	0.84	0.40	0.54

On LargeDS (4,076 apps), the MAS labels 1,395 repackaged apps as malware (34.22%), i.e., those invoking at least one additional sensitive API. However, performance drops sharply: F_1 decreases from 0.89 (SmallDS) to 0.54, indicating substantially lower accuracy at scale. This degradation motivates further analysis (RQ2 and RQ3). A logistic regression confirms the effect is statistically significant ($p < 0.001$), rejecting the null hypothesis and linking accuracy to app similarity[12].

4.2 Evaluation According to Similarity Benchmark.

Figure 2 shows similarity distributions for SmallDS and LargeDS. SmallDS has an average similarity of 0.89 (median 0.99, SD 0.25), while LargeDS averages 0.90 (median 0.98, SD 0.18), indicating most samples are highly similar. We analyze the impact of similarity on MAS accuracy using logistic regression, excluding true negatives. The null hypothesis (H_0)—that similarity does not affect accuracy—is rejected ($p = 2.22 \cdot 10^{-16}$), confirming a significant relationship. Using K-Means, we cluster LargeDS into ten groups by similarity. Results (Table 4) show highest accuracy (70.40%) at moderate similarity (0.67),

while highly similar samples (0.99, 1,248 apps) achieve only 26.68% accuracy. Overall, lower similarity tends to improve MAS performance, explaining its reduced accuracy on LargeDS.

Table 4: Characteristics of the ten similarity-based clusters. The table shows the average similarity score, number of samples, correct answers, and the corresponding percentage of correct classification for each cluster.

cId	Sim. Score	Samples	Correct Answers	%
1	0.09	93	46	49.46
2	0.55	175	106	60.57
3	0.67	125	88	70.40
4	0.79	165	90	54.55
5	0.87	261	77	29.50
6	0.90	230	121	52.61
7	0.94	141	46	32.62
8	0.97	140	59	42.14
9	0.98	398	95	23.87
10	0.99	1248	333	26.68

4.3 Assessment Based on Malware Family

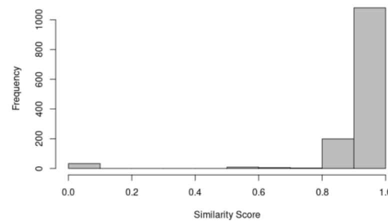
Similarity alone does not explain MAS's low performance on LargeDS; malware family diversity also plays a role. LargeDS spans 116 families, with Gappusin (1,337), Revmob (207), Dowgin (183), and Airpush (120) accounting for 63.79% of VT-labeled malware. In contrast, SmallDS is dominated by Kuguo, Dowgin, and Youmi, with minimal representation of Gappusin and none of Revmob.

Evaluating MAS on Gappusin and Revmob (Table 5), we find substantial misclassification: 87.5% of Gappusin (1,170 samples) and 44.44% of Revmob (92 samples) are missed. Excluding these families improves recall to 0.72, though still below prior results.

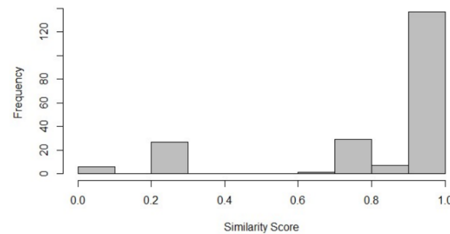
Table 5: Confusion matrix of the MAS when considering only the samples from the Gappusin and Revmob families in the LargeDS.

Actual Condition	Predicted Condition	
	Benign	Malware
Benign (0)	TN (0)	FP (0)
Gappusin (1,337)	FN (1,170)	TP (167)
Revmob (207)	FN (92)	TP (115)

We further analyze Gappusin and Revmob due to their impact on results. Both families show high similarity between original and repackaged apps (Gappusin: avg 0.94, median 0.99; Revmob: avg 0.81, median 0.91), as shown in Figure 2. Manual inspection of 30 samples per family reveals that Gappusin typically makes minimal changes: no new permissions, only minor Manifest edits (e.g., package or activity name). In 29/30 cases, it modifies the `onReceive(Context, Intent)` method in `AdReciver` to download a different `data.apk`, often via an added helper method. Since these changes do not introduce new sensitive API calls, MAS fails to detect them.



(a) Similarity score distribution for the Gappusin family.



(b) Similarity score distribution for the Revmob family.

Figure 2: Histogram of similarity scores for the Gappusin and Revmob families. Both families show high similarity between original and repackaged versions, contributing to the low detection rate.

Our analysis also shows recurring deletions in repackaged apps. In 20/30 Gappusin samples, method `void b(Context)` from `com.game.a`—which heavily uses reflection—is removed. This may simplify downloading a different `data.apk` or help evade antivirus detection, as reflection and `DexClassLoader` can enable dynamic code injection [13]. Since no new sensitive API calls are introduced, MAS fails to detect such modifications.

For Revmob, we analyzed 30 undetected samples. Like Gappusin, none modify the Manifest or add sensitive API calls, complicating detection. However, all include a native ".so" library in the `lib` directory. These shared objects can embed malicious C/C++ code and enable dynamic code injection via `System.loadLibrary()` or `System.load()` [13, 14]. Once loaded, they interact with Java through JNI, bypassing MAS.

All samples load `libgame.so` via `System.loadLibrary("game")` in the `mainActivity` static initializer.

The `libgame.so` file contains native code compiled from C or C++, which is loaded at runtime and connected to the app. Because this is low-level machine code, it is not straightforward to analyze. Still, we observed that each file includes the function, which is used by JNI to link Java code with native methods. When we inspected the hexadecimal representation of these ".so" files, we noticed clear differences in both size and content between the original and repackaged versions. This indicates that important modifications may exist within the native `libgame.so` component, which MAS does not currently capture.

5. Discussion

This article will provide answers to the research questions in the following article.

5.1 Answers to the Research Questions

- **RQ1 (Performance on LargeDS).** The MAS accuracy reported in prior studies [2, 3] does not generalize to larger datasets. On SmallDS we reproduced 0.89 accuracy, but on LargeDS (4,076 pairs) accuracy dropped to 0.54.
- **RQ2 (Similarity).** There is an association between original/repackaged app similarity and MAS's ability to correctly classify malware. Similarity explains the low performance on LargeDS.
- **RQ3 (Malware Families).** Certain families cause most false negatives. We investigated Gappusin and Revmob (adware). Reverse engineering 60 samples confirmed MAS cannot identify their change patterns. Their prevalence in LargeDS further explains poor performance.

5.2 Implications

Contrary to previous work, our results provide a systematic understanding of MAS strengths and limitations. This is the first empirical evaluation of MAS using VT as ground truth, enabling standard accuracy metrics (precision, recall, F_1). Prior studies assumed all repackaged apps were malware; our VT triangulation shows this is false. MAS achieves $F_1 = 0.89$ on SmallDS but drops to 0.54 on LargeDS. This observation aligns with recent findings that detection approaches often fail to generalize under realistic conditions, particularly in the presence of repackaged or highly similar applications. We also identified families responsible for many false negatives. Key takeaways:

- Negative result: MAS for malware detection has a much higher false negative rate than previously reported.
- Future directions: Researchers must advance MAS by identifying change patterns in repackaged apps and mining sensitive native API calls. Malware increasingly uses JNI to hide code in the native layer, which current sandbox mining overlooks.

5.3 Threats to Validity

External validity: While prior work used only 102 pairs, our larger dataset (4,076 pairs) improves representativeness with diverse families and similarity levels. However, findings are limited to Android repackaged malware.

Conclusion validity: MAS assumes benign originals. We verified this with VT; seven SmallDS originals were flagged as malicious (noting VT results may change over time [7]). LargeDS includes only pairs with VT-benign originals.

Construct validity: We use standard metrics (precision, recall, F_1) based on true/false positives. A repackaged app is labeled malware if flagged by at least two VT engines; alternative thresholds yield similar F_1 (0.53),. Excluding malicious originals may introduce bias but aligns with MAS assumptions.

6. Conclusions

To better understand the strengths and limitations of MAS for detecting repackaged malware, we reproduced earlier studies [2, 3] using a more diverse dataset. The results show a clear drop in performance, with the F_1 score decreasing from 0.89 to 0.54. This decline is largely linked to the presence of malware families such as Gappusin and Revmob, which are often misclassified by MAS. A closer look through

reverse engineering helps explain why. In the case of Gappusin, the repackaged apps rely on reflection to download and install external APKs, without introducing new sensitive API calls. Revmob, on the other hand, makes use of JNI to execute malicious behavior within native code, carrying out low-level operations that are difficult to detect. In both cases, these techniques allow the malware to bypass MAS. Overall, these findings point to clear limitations in MAS and suggest that it should be combined with other approaches to improve the effectiveness of malware detection.

References

[1] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Trans. Software Eng.*, 47(4):676–693, 2021. doi: 10.1109/TSE.2019.2901679.

[2] Lingfeng Bao, Tien-Duy B. Le, and David Lo. Mining sandboxes: Are we there yet? In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018*, pages 445–455. IEEE Computer Society, 2018. doi: 10.1109/SANER.2018.8330231.

[3] Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro. Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *J. Syst. Softw.*, 183:111092, 2022. doi: 10.1016/j.jss.2021.111092.

[4] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Companion Volume*, pages 23–26. IEEE Computer Society, 2017. doi: 10.1109/ICSE-C.2017.8.

[5] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 37–48. ACM, 2016. doi: 10.1145/2884781.2884782.

[6] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016*, pages 468–471. ACM, 2016. doi: 10.1145/2901739.2903508.

[7] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium, USENIX Security 2020*, pages 2361–2378. USENIX Association, 2020.

[8] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - fine-grained policy enforcement for untrusted android applications. In *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013*, volume 8247 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2013. doi: 10.1007/978-3-642-54568-9_14.

[9] Francisco Handrick da Costa, Ismael Medeiros, Pedro Costa, Thales Menezes, Marcos Vinícius, Rodrigo Bonifácio, and Edna Dias Canedo. Droidxp: A benchmark for supporting the research on

mining android sandboxes. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020*, pages 143–148. IEEE, 2020. doi: 10.1109/SCAM51674.2020.00021.

[10] Y. Pan, X. Ge, C. Fang, and Y. Fan. A systematic literature review of android malware detection using static analysis. *IEEE Access*, 8:116363–116379, 2020.

[11] J. Qiu. Survey on android malware detection techniques. *IEEE Access*, 8, 2020.

[12] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu. A review of android malware detection approaches based on machine learning. *IEEE Access*, 8:124579–124607, 2020.

[13] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab 'n run: Secure and practical dynamic code loading for android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015*, pages 201–210. ACM, 2015. doi: 10.1145/2818000.2818042.

[14] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. On tracking information flows through JNI in android applications. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, pages 180–191. IEEE Computer Society, 2014. doi: 10.1109/DSN.2014.30.